"A Tale of Two Provers" :

A Comparison of Dependent Haskell and F*

Xinyue Zhang (xzhang@brynmawr.edu)

Advised by Richard A. Eisenberg

February, 2018

1 Introduction

> *"Testing can show the presence of errors, but not their absence."*
>
> – E. W. Dijkstra

Programming Languages are powerful tools for designing and building complex computer programs to perform various tasks. Yet complex programs often have bugs that prevent them to function as expected. Programmers want to not only make sure that the programs written indeed perform as intended but also guarantee that these programs are safe and secure. Through testing, it's possible to find incorrect behaviors for a certain number of inputs tested, assuming that the correctness of output is easy to check. On the other hand, programmers can prove the correctness of the program regardless of its inputs. Therefore, finding ways to help programming languages to become both verified and expressive is an exciting direction of research.

Program verification is a technique, which uses formal mathematical methods to prove the correctness of a software program or system, with respect to a formally defined or inferred specification. It can detect both program bugs and system vulnerabilities at compile time.

Dependent type is a type whose definition is predicated upon a dynamic value. As such, dependently-typed programming enables programmers to express more detailed specifications with types and perform more powerful verifications through type checking. "Dependently typed programs are, by their nature, proof carrying code" [13]. Fully dependently typed languages, like Coq [3], Agda [15], Idris [2] and Dafny [11] have been studied for a long time and are relatively well understood. On the other hand, functional languages, providing a good foundation with its connection to mathematical functions, are relatively new in program verification. Researchers on functional programming languages, like Haskell and ML, gradually gain more insights in providing verification supports using Dependent types. Dependent Haskell and F* are two such languages, each presents a unique and effective program verification technique.

2 Motivation

Dependent Haskell and F* are similar in that they both aim towards a verification-based yet general-purpose programming language, but they are noticeably different in their unique approaches to verify potential divergent functions. Non-terminating functions are very common in practical programming languages. Thus, being able to express and reason about programs that potentially diverge is an ongoing research direction.

Dependent Haskell doesn't require totality checking at compile time. As a partial language, Dependent Haskell has proofs that diverges or errors, and contrary to most dependently typed languages, "verifies" proof termination at run-time. The "run-time termination checking" essentially just runs the implemented algorithms without termination checking ; Haskellers either wait until program actually terminates or manually kills the process for divergent functions. On the other hand, F* categorizes non-termination as a programming effect, and designs a full lattice of monadic effects for program verification.

In 2017, Morris and Eisenberg acknowledged in Constrained Type Families that the current Haskell type system inevitably accepts some apparently erroneous definitions, as Haskellś type families equivocally assumes totality [14]. Working in F* for a semester, we find its approach towards non-terminating functions promising and want to know if it can give us insights into achieving compile-time termination checking while still keeping the completeness of proofs.

In this undergraduate thesis, we aim to introduce the two verification-oriented programming languages, Dependent Haskell and F*, present their unique type system designs with concrete examples, and give a detailed comparison from a combination of theoretical and practical standpoints. We expect our research to setup a solid foundation for potential collaborations across the two communities, and to possibly propose promising future research directions in program verifications for the general programming languages community.

3 Approach and Novelty

To gain a better understanding of the language design decisions and verification techniques, We first carefully study the literatures of each language. In general, our assessment of Dependent Haskell and F* is based on implementing and proving a wide spectrum of algorithms in both languages. These programs are written so that we could have a much more in-depth experience with both languages and explore their features more thoroughly. To begin with, in both Dependent Haskell and F*, we implement the data type of a length-indexed vector and verify various functions that support types of finite length vectors from Haskell's List module, including length, head, tail, init, last, snoc, reverse, and, or, null etc. We then focus on the verification for various sorting algorithms, such as mergesort and quicksort, and examine the capability of proving divergent functions, as in the proof of peano division with zero divisor and the equivalence of multi-step and big-step evaluators in Simply Typed Lambda Calculus. We expect to structure the comparison mainly focusing on two aspects : general language design and support measured from syntactic verbosity and library/documentation resources ; verification techniques focusing on proof

complexity and proof completeness when verifying potential non-terminating functions. As far as we know, we are the first researcher to directly compare program verification techniques between Dependent Haskell and F*.

4 Background

4.1 From Imperative to functional programming

Most programmers come from a Java or C/C++ background. These languages are designed to primarily support imperative, or procedural programming. In the imperative style of programming, a developer write code that details each step for computers to execute. These imperative programs often consists of stateful statements that modifies global states when executed. Functional programming, as its name suggests, is a style of programming that takes a pure functional approach. In functional programming, a developer models the problem as a set of functions to be executed, detailing the input and output from the each function. Functional programs are usually executed by evaluating expressions and typically avoids mutating program states. Common functional programming languages are Haskell, ML, Scala, and Lisp.

In functional programming, functions are first-class citizens, i.e. they are treated like any other values and can be passed as arguments to other functions or be returned as a result of a function. The functions that take functional arguments are called Higher-order Functions [1].

4.2 Program Specifications with dependent, refinement types

To verify a program, we first need to express its specification. Dependent Haskell, specifically adopts dependent types, which are types indexed by arbitrary total expressions, with type-level computation defining an equivalence relation on types. In addition to the predicated dependent types, refinement type is another way to specify a program. A refinement of a type t is a type `x:t{`ϕ`}` inhabited by expressions `e : Tot t` that additionally validate the logical formula ϕ`[e/x]`. Refinement types reduces code redundancy and syntactic complexity through subtyping [18].

For example, consider a length indexed list `l` and let `n` represents the length of the list. Then `l` is specified through dependent type as `l:list a n`, and through refinement type as `l:list a{len l = n}`.

4.3 Program Verification

In functional programming and type theory, an algebraic data type (ADT) is a composite type that combines other types. GADT, short for Generalized Algebraic Data Types, allows term-level patterns to be matched to refine type level statements.

For example, the definition of list is defined as follows :

1. https ://wiki.haskell.org/Functional_programming

```haskell
-- A parametric ADT that is not a GADT
data List a = Nil | Cons a (List a)

-- A GADT
data List a where
  Nil: List a
  Cons: a -> List a -> List a
```

Similarly, the recursive definition of a natural number `Nat` is :

```haskell
data Nat where
  Zero :: Nat
  Succ :: Nat -> Nat
```

It follows that, using dependent types, we can specify the length of a list in type by parameterizing it as natural numbers, thus introducing the dependently typed `Vec` definition :

```haskell
data Vec :: Nat -> Type -> Type where
  Nil  :: Vec Zero a
  (:>) :: a -> Vec n a -> Vec (Succ n) a
infixr 5 :>
-- enable printing out for debugging
deriving instance Show a => Show (Vec n a)
```

Finally, we introduce the concept of typeclass. Typeclass can be understood as an interface that classifies a group of behaviors. Typeclass contains many types that all support and implement the behavior their typeclass describes [2]. For example, `Ord` is a typeclass for all types that have an ordering.

With the parametrized dependently-typed GADT `Vec` and the typeclass `Ord`, we begin with a simple example on how program verification works in Haskell.

Specifically we define the `insert` function to insert an element to a vector.

```haskell
insert :: Ord a => a -> Vec n a -> Vec (Succ n) a
insert num Nil     = num :> Nil
insert num (x :> xs) = case (num > x) of
  True  -> x :> (insert num xs)
  False -> num :> x :> xs
```

2. http ://learnyouahaskell.com/types-and-typeclasses

Similar to writing test cases, in program verification, we focus on the specification of each program. For insert, we know that after inserting a new element, the length of the list is incremented by 1. Therefore, we define the type for function `insert` as `Ord a => a -> Vec n a -> Vec (Succ n) a`. The actual proof is then simple from induction.

To prove some more complexed properties, for example concatenating two `Vec`s, we need to verify the specification such that the resulting `Vec` has length as the sum of the two concatenated `Vec`s. To perform the addition operation in type level, we need to introduce what's known as a type family. From Haskell Wiki on Type Family, it is a partial function at the type level, such that when applied to parameters yields a type. Type families give us the ability to compute over types. For example, to add two type-level `Nat`s together, we define the type family as :

```
type family (a :: Nat) + (b :: Nat) :: Nat where
  Zero   + b = b
  Succ a + b = Succ (a + b)
infixl 6 +
```

Thus, `concat` can be easily implemented and verified as :

```
concat :: Vec n a -> Vec m a -> Vec (n + m) a
concat Nil v2     = v2
concat (x :> xs) v2 = x :> (concat xs v2)
```

Besides programming and proving, we also want to reason about two main properties of the language's type system, as soundness and completeness. We call a system sound if every property that can be proved in the system is logically valid with respect to the semantics of the system. On the other hand, we call the system complete if every logically valid property with respect to the semantics of the system can be proved by the system. Generally, we strive to make a type system more complete without sacrificing the soundness of each proof.

4.4 Proof Automation

Automated theorem proving is a subfield of mathematical logic that proves mathematical theorems through computer programs. Satisfiability Modulo Theories (SMT) solvers provide a good foundation for highly automated programs [12].

Satisfiability Modulo Theories (SMT) problems are a set of decision problems that generalize boolean satisfiability (SAT) with arithmetic, fixed-sized bit-vectors, arrays, and other related first-order theories. Z3 is a new and efficient SMT Solver introduced by Microsoft Research that is widely adopted in program verifications [4]. The F* language achieves semi-automation through a combination of SMT solving using Z3 and user-provided proof terms.

4.5 Programming Effects

In practical programming, program often doesn't just return a value but also has side effects, modifying states outside their scopes or interacting with its calling functions or the outside world. An effect system is a formal system which describes the computational effects of computer programs, such as I/O, error handling, and divergence etc. An effect system can be used to provide a compile-time check of the possible effects of the program.

In most functional programming languages, one common way to express programing effects is by simulating with monads. Through monads, programmers can structure computations in terms of values and sequences of computations using those values. Monads keep the language itself pure.

Aimed at a language that servers as interactive theorem prover, general purpose programming language and semi-automated verifier, F* categorizes programming effects into a lattice of monads [18]. Specifically, the `PURE` monad isolates all pure computations, whose soundness must be guaranteed from a compile-time semantic termination check based on a well-founded order. A well-founded ordering must ensures that each iteration of the recursive function is applied to a strictly smaller domain. F* on default provides four built-in ordering, but also allows specifying explicit decreasing metrics on function parameters. Besides the core `Pure` monad, F* provides the primitive `DIV` monad that models all computations that potentially diverges, the `STATE` monad for all stateful computations, `EXN` monad for programs that may raise exceptions, and finally the `ALL` monads that categorizes all programming effects [18].

5 Related Work

5.1 Language Comparison

There are many programming languages in use today and new ones are created every year. Selecting a language among them is hard for general programmers and even experienced computer scientists. Various work has been done to compare programming languages. Through comparison, researchers find advantages and limitations in each and evaluate to propose insights for future research [7].

In the realm of programming languages, there are general-purposed programming languages like Pascal, Haskell and C/C++, interactive proof assistants like Coq, Agda and Isabelle/HOL, and SMT-based semi-automated program verification tools like Dafny, Vampire [10] and WhyML [18]. Wiedijk compared a list of proof assistants on mathematical theorems, focusing on their strength of logic and level of automation [19] [20]. Feuer and Gehani edited papers comparing and assessing general purposed languages Ada, C and Pascal [7]. Filliâtre compared Vampire and Dafny in proving steps of the progress proof [16] [8].

As the needs grow for general purpose yet verification oriented programming languages, four functional programming languages are recently invented, namely Dependent Haskell, Liquid Haskell, F* and Idris. As an ongoing field of research, to my best knowledge, these

four communities generally conduct research separately and have rarely explored each other's approaches in-depth.

At the 45th ACM POPL Student Research Competition, Xu and Zhang presented their collaborated work on the comparison among three programing verification techniques in Dependent Haskell, Liquid Haskell and F* [21]. As an exploratory research, they found promising aspects in each language and proposed more detailed directions for further comparisons. In this thesis, we aim to provide an in-depth comparison between Dependent Haskell and F*, with a focus on their different verification approaches towards non-terminating functions. Through in-depth assessments of the verification techniques in both languages, we aim to find the more desirable approach and to recommend future directions for research in termination checking.

5.2 Haskell and Dependent Haskell

Haskell is a strongly-typed purely functional programming language. It features a type system with type inference and lazy evaluation. For a long time, Haskell researchers strive to add dependent types into Haskell, as dependent types introduce much more precise expressions of program specifications. Although Haskell hasn't yet supported dependent types in full, several extensions on its current type system have made exceptional progress. In 2012, Eisenberg and Weirich introduced the `singletons` library that essentially achieves dependently-typed programming in Haskell. Singleton types allow Haskell programmers to enforce rich constraints among the types in their programs using dependently typed techniques [6]. It also allows promotion of term-level functions to type-level equivalents [6]. Later, Gundry and Eisenberg both conducted research on dependently-typed Haskell using GADTs and type families. Gundry contributes a rebuilt algorithm for first-order unification and Hindley-Milner type inference, which is applied in the constrained solver underlying the elaboration algorithm behind Gundry's implementation [9]. Eisenberg, extending Gundry's work, goes one step further to blur the distinction for type and kind levels, unifying expressions and types in Dependent Haskell [5]. Without compile-time termination checking, Dependent Haskell only guarantees correctness of fully evaluated proofs, i.e. there can only be values of the correct types. Though erroneous proofs like `1 :~: 2`, a propositional equality GADT representing $1 = 2$, might be verified, they will never be fully evaluated down to values.

5.3 F*

F* is an ML-like functional programming language aimed at program verification [18]. In 2016, Swamy et al. introduced the current, completely redesigned version of F*, a language that works as a powerful and interactive proof assistant, a general-purpose, verification-oriented and effectful programming language, and an SMT-based semi-automated program verification tool. F* is a dependently typed, higher-order and call-by-value language with a lattice of primitive effects. These monadic effects can be specified by the programmer and is used by F* to discharge verifications using both SMT solver and manual proofs [18]. In addition, it uses a combination of dependent types and refinement types for program ve-

rification and supports termination checking based on a metric of well-founded decreasing order to ensure program consistency. Dependently typed F* is further enhanced by the introduction of Dijkstra Monads to perform program verification on effectful code beyond the primitive effects by Ahman et al [1].

6 Current Results

6.1 Efforts To Date

As summarized in Section 3, Approach and Novelty, we have implemented various functions that supports types of finite length vectors from the Haskell List module, two machine-checked sorting algorithms – MergeSort and QuickSort, and a proof on peano division with zero divisor.

In terms of general language design and support, we conclude that Dependent Haskell is syntactic verbose. In Dependent Haskell, programmers are expected to implement the same logic twice. The logic, for example the addition of two natural numbers, need to be specified once in term level as usual but again in type level using type families. Also, Dependent Haskell requires the use of singletons, which captures the same logic of a GADT, but promoted in the type level. On the other hand, F* is syntactically concise. It supports both type-level specifications through dependent types and type-level computations through refinement types. Also, through its monadic return typing rule, F* can automatically enrich the types of a function and reflect the existing term-level implementation into logic for reasoning.

As Haskell is a mature, industrial-strength programming language, Dependent Haskell, extending Haskell with dependent types, inherits all the resources . Specifically, Haskell provides a central package archive called Hackage where packages are introduced, maintained, and supported. In addition, Hoogle is a holistic search engine for Haskell APIs, which provides advanced queries through function name and even approximate type signatures. However, as a living language that is continuously developing, F* is still in lack of many resources. Most of the documentations are through its tutorial and the official FStar Github repository.

Regarding to the verification techniques, Dependent Haskell is purely based on an extension of Martin-Löf Type Theory, doesn't prove any proof interaction or automation. Unlike most verifiers, it chooses to avoid compile-time termination checking but supports most general programming effects, like IO and States through monads. Verifications in Dependent Haskell is guaranteed both sound and complete, but could get really tedious even for relatively simple sorting algorithms.

F*, on the other hand, provides a flexible combination of SMT-based proof automation and manually constructed proofs. In addition, F* can use SMT solver, such as Z3, to match SMT patterns provided in the manually provided lemmas to assist with the constructive proofs. F* supports theorem proving through either enriching function types(intrinsic style) or by writing separate lemmas(extrinsic style) [18]. F* is an effectful programming language that categorize effects into a lattice of monads. Specifically it requires explicit

annotation on effects and enforces termination checking on all `Tot` functions.

To find out the expressiveness of the F* language, especially regarding to the verifications on non-terminating functions, we begin by exploring the peano division example. Peano naturals are a simple way to encode natural numbers, with the base value `Zero` and the `Succ` function. For example, natural number 0 is expressed as `Zero` and 1 is expressed as `Succ Zero` etc. In our example, we want to show that the quotient of a product of two natural numbers a, b and b is equal to a. As an educational example, we removed the quantification on divisor to allow zero divisors in our definition of `` `div` `` and try to prove the valid lemma on division with nonzero divisors. Dependent Haskell, which checks program termination at "run-time" can successfully verify the module. However, F*, based on the relative complete Hoare Logic, doesn't support extrinsic proofs on non-terminating functions. Through it allows proving divergent programs intrinsically through type enrichment, the approach inevitably requires programmers to know the exact reason of non-termination, which is not very practical in real-world programming.

6.2 Future Plans

We are in the progress of proving the equivalence of multi-step evaluator and big-step evaluator in Simply Typed Lambda Calculus. Simply Typed Lambda-Calculus, or STLC, is a tiny core calculus incorporating the key concepts of functional abstraction, that is incorporated in most real-world programming languages [17]. As STLC has enough structural complexity to illustrate interesting theoretical properties, it is commonly used to evaluate programming languages. STLC is based on a collection of base types, and for our purposes, we chose to study the simply typed lambda-calculus with booleans, which contains the following terms :

$$
\begin{array}{rcl}
term & ::= & x \\
& | & \lambda x : T \cdot t \\
& | & t\, t \\
& | & True \\
& | & False \\
& | & if\ t_1\ then\ t_2\ else\ t_3
\end{array}
$$

We also define values as the term where reduction stops, i.e. cannot evaluate/step further. Specifically in our case, `True` and `False` are obviously values, and we also define the abstraction term $\lambda x : T \cdot t$ as values as reduction stops at abstractions.

Specifically, we first implemented the simple step function, following the operational semantics of the evaluation rules, that evaluate one step at a time. We then extends it to what's called a multi-step evaluation function that calls step function whenever the evaluated term is not a value. Finally, we implemented the big-step evaluation function that directly evaluates down to the final value. From type theory, we know that in Simply Typed Lambda Calculus, both functions evaluated down to values and are indeed equivalent. However, as there isn't a clear decreasing metric, we aren't sure if F* could be able to prove their totality. Hence, we are interested to explore if F*'s effect system is complete

enough to handle this case. If so, we want to move on to examine what Haskell could adopt ; otherwise if not, we will explore in detail the advantages and limitations of the current effect systems and conclude a recommendation for the future directions.

Références

[1] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17. ACM, 2017.

[2] Edwin Brady. Idris, a general-purpose dependently typed programming language : Design and implementation. *J. Funct. Prog.*, 23, 2013.

[3] Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] Richard A. Eisenberg. *Dependent Types in Haskell : Theory and Practice*. PhD thesis, University of Pennsylvania, 2016.

[6] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

[7] Narain Gehani. *Comparing and assessing programming languages : Ada, C, and Pascal*. Prentice Hall, 1984.

[8] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. Automating proof steps of progress proofs : Comparing vampire and dafny. In *Vampire@ IJCAR*, pages 33–45, 2016.

[9] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[10] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[11] K. Rustan M. Leino. Dafny : An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'16, pages 348–370. Springer Berlin Heidelberg, 2010.

[12] K Rustan M Leino. Automating induction with an smt solver. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 315–331. Springer, 2012.

[13] James McKinna. Why dependent types matter. In *ACM Sigplan Notices*, volume 41, pages 1–1. ACM, 2006.

[14] J Garrett Morris and Richard A Eisenberg. Constrained type families. In *Proceedings of the ACM on Programming Languages*, ICFP '17. ACM, 2017.

[15] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[16] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[17] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage : http ://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[18] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[19] Freek Wiedijk. Comparing mathematical provers. In *International Conference on Mathematical Knowledge Management*, pages 188–202. Springer, 2003.

[20] Freek Wiedijk. *The seventeen provers of the world : Foreword by Dana S. Scott*, volume 3600. Springer, 2006.

[21] Rachel Xu and Xinyue Zhang. Comparisons among three programming verification techniques in dependent haskell, liquid haskell and f*. In *Proceedings of the 45th Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '18. ACM, 2018.

I really like how you start your paper with a quote. Both the quote and the first paragraph provide a clear and strong motivation. I think it would be better if you can give a brief outline of what you are going to write in this paper so readers will know what to look forwards to.

I also like your mixture of explanation and code. It makes everything easier to understand.

One suggestion would be to rethink about your structure. I feel the logic of your sections is not so smooth. I think a outline in introduction can help with that. Also add more transactions between sections may also help.

Overall, I think you paper does a good job in explaining the field and your work to someone who is not familiar with Haskell.